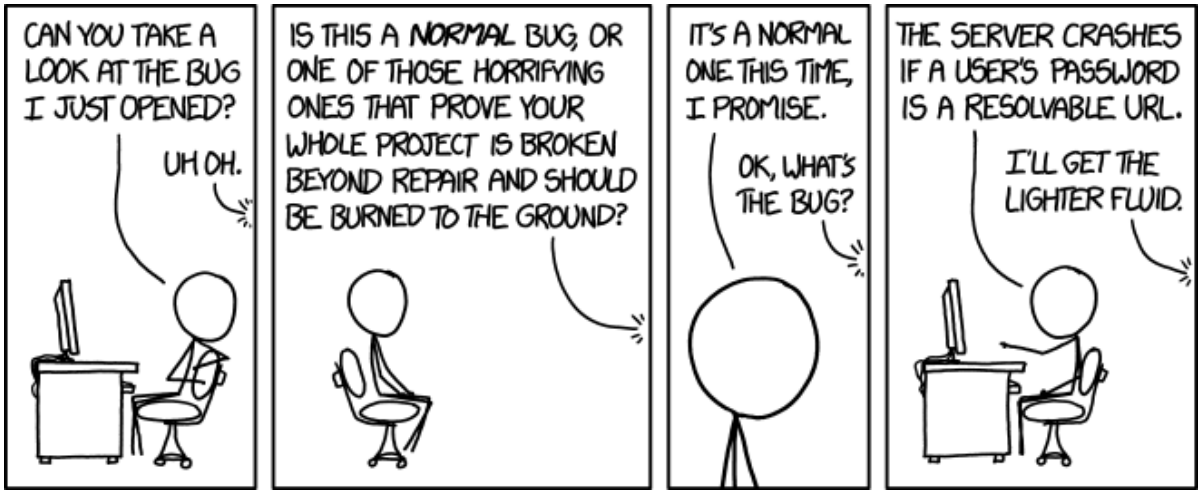
TP0 : Tests

Objectifs du TP :

- (Re)voir quelques points de syntaxe simples en C.
- Lire, comprendre et tester un code.
- Souligner l'importance d'un code clair et des tests afférents.



https://xkcd.com/1700/

Un fichier TP0_sujet.c est mis à votre disposition dans l'espace partagé réservé à la classe. Vous y trouverez les fonctions ci-après mentionnées. Pour chacune d'entre elles il est demandé de :

- Inférer une spécification précise de la fonction via la lecture du code et des tests. On pourra écrire des fonctions de test dédiées, on conservera les tests effectués et on cherchera à les rendre exhaustifs.
- Renommer la fonction de sorte à ce que son nom ait une pertinence vis-à-vis de sa spécification.
- Répondre aux questions spécifiques ci-dessous.

1. Fonction triangle.
   a) Pourquoi le premier test sur les valeurs de i, j, k est-il important ?
   b) Modifier les affichages de manière pertinente.
2. Fonction divergence.
   a) Expliquer quel résultat bien connu permet de garantir la terminaison de cette fonction.
3. Fonction a000045.
   a) Rappeler les problèmes que pose une fonction récursive répondant à la même spécification que a000045.
   b) Renommer les variables de cette fonction de manière pertinente.
4. Fonction chaines.
   a) Réimplémenter à la main la fonction strlen.
   b) Modifier les affichages de la fonction chaines de manière pertinente.

5. Fonction soleil. Types entiers en C.

   Le module stdint.h définit plusieurs types d'entiers qui diffèrent selon leur caractère signé ou non et leur largeur (nombre de bits utilisés pour le codage). Plus précisément :

Type	Entier signé	Sur _ bits	Intervalle représenté
uint8_t	non	8	?
int8_t	oui	8	?
uint16_t	non	16	$[\![0, 65535]\!]$
int16_t	oui	16	$[\![-32768, 32767]\!]$
uint32_t	non	32	?
int32_t	oui	32	?
uint64_t	non	64	?
int64_t	oui	64	?

   L'avantage de ces types est qu'ils sont clairement définis. Ils sont indépendants de la machine ou du compilateur. Ce n'est pas le cas par exemple du type int qui code des entiers sur un nombre de bits différent selon qu'on est en présence d'une architecture 64 ou 32 bits. Le plus souvent (sur les architectures 64-bits donc), le type int correspond en fait au type int32_t. Lorsque l'utilisation d'une taille précise n'est pas nécessaire, on continuera à utiliser les types int et unsigned int.

   a) Compléter le tableau ci-dessus.

   b) Décrire chacun des cas de la fonction soleil afin d'en expliciter la compréhension.

6. Fonction extraction. Types constants et opérations bits à bits.

   Cette fonction est l'occasion d'introduire deux notions au programme que nous avons peu vues :

   - Les variables BLOCK_SIZE, BASE et MASK sont de type const int. Le mot clé const devant un type t permet de créer un nouveau type const t et la valeur d'une variable ayant ce type n'est accessible qu'en lecture : on ne peut pas la modifier. Ainsi, ce code provoque une erreur :

```
const float pi = 3.14;

int main(){pi = 5;}
```

     Les types const sont utiles lorsqu'on souhaite protéger la valeur d'une variable, c'est-à-dire interdire sa modification (que ce soit par étourderie ou malveillance).

     Remarque importante en ce qui concerne la définition de tableaux statiques : Nous avons vu que la taille d'un tableau statique doit être connue à la compilation. Une variable de type const int reste une variable (donc non connue à la compilation), aussi écrire :

```
const int taille = 100;
int tableau[taille];
```

     est interdit (et selon votre compilateur, vous obtiendrez même une erreur). On doit donc écrire :

```
const int taille = 100;
int tableau[100];
```

     La variable taille peut être utilisée dans le code (pour des bornes de boucles permettant de traiter le tableau par exemple) mais pas dans la définition du tableau même. Bien évidemment cela pose le problème suivant : si on veut changer la taille du tableau, il faudra faire deux modifications ! C'est néanmoins cette façon de faire qui est imposée par le programme de

MPI. Pour information, la bonne façon de faire (hors programme) est d'utiliser la commande de pré-processeur #define que nous avons vu brièvement l'année dernière :

```
#define taille 100
int tableau[taille];
```

- La fonction extraction utilise des opérateurs bit à bit. Un tel opérateur agit sur les représentations binaires de ses arguments. Plus précisément on a :

Opération	Syntaxe	Exemple
Conjonction	&	$10\&9 = 8$
Disjonction	|	$10\|9 = 11$
Ou exclusif	^	$10\hat{}9 = 3$
Décalage à gauche	$\ll$	$3 \ll 2 = 12$
Décalage à droite	$\gg$	$42 \gg 2 = 10$

Attention à ne pas confondre la syntaxe des opérateurs logiques "ou" et "et" avec la syntaxe des opérateurs bit à bit "ou" et "et" !

a) Expliquer le résultat obtenu sur chacun des exemples dans le tableau ci-dessus.

b) Si n et i appartiennent à $\mathbb{N}$, à quoi revient l'opération $n \ll i$? Et l'opération $n \gg i$?

c) Le langage C dispose également d'un opérateur de négation bit à bit ($\sim$). Pourquoi ne peut-on pas immédiatement savoir ce que vaut ~ 10 ?

7. Fonction kramp.

   a) A quoi faut-il faire attention lors du test de cette fonction ?

8. Fonction elucide.

   a) Ecrire une fonction qui étend elucide aux entiers négatifs.

9. Fonction etienne.

   a) Pourquoi fait-on de x, y et z des arguments de la fonction sous_etienne ?

   b) Justifier la correction de la fonction etienne.

Bonus : Les noms des fonctions etienne, elucide, kramp et a000045 n'ont pas été choisis au hasard... voyez vous à quoi ils font référence ?